24

Interpolation

A problem that often arises in data analysis is *interpolation*, that is, estimating the value of a function between points at which the function is known. This chapter presents several simple interpolation examples using the built-in IDL interpolation functions.

24.1 IDL Commands and Keywords

The following built-in IDL functions can be used to interpolate data:

- `INTERPOL` function
- `BILINEAR` function
- `INTERPOLATE` function
- `TRIANGULATE` function
- `TRIGRID` function

24.2 Background

Given a function that is tabulated at a finite set of points, *interpolation* is the problem of estimating the value of the function at locations *between* the tabulated points. *Extrapolation* is the problem of estimating the value of the function *outside* the range of tabulated points. To interpolate or extrapolate, the tabulated values are used to construct an *interpolating function*. The interpolating function is often a piecewise polynomial of relatively low order, typically linear, quadratic, or cubic, although other kinds of functions can be used. In order to be considered interpolation, as opposed to *curve fitting*, the interpolating function should pass *exactly* through the tabulated points.

IDL includes several built-in functions to do interpolation using various kinds of interpolating functions. These include `INTERPOL` and `INTERPOLATE`.

24.3 1-D Interpolation

The IDL function `INTERPOL` can do several different kinds of one-dimensional interpolation, specifically *linear*, *quadratic*, and *cubic spline* interpolation.

Here is a quick demonstration of how to use INTERPOL. Annotated versions of the resulting graphs are plotted in Figure 24.1.

```
IDL> x = findgen(6)
IDL> y = [0.1, 0.9, 0.2, 0.8, 0.3, 0.7]
IDL> xx = 5.0*findgen(26)/25
IDL> yy = interpol(y, x, xx)
IDL> plot, x, y, psym = -4, symsize = 2
IDL> oplot, xx, yy, psym = -1
```

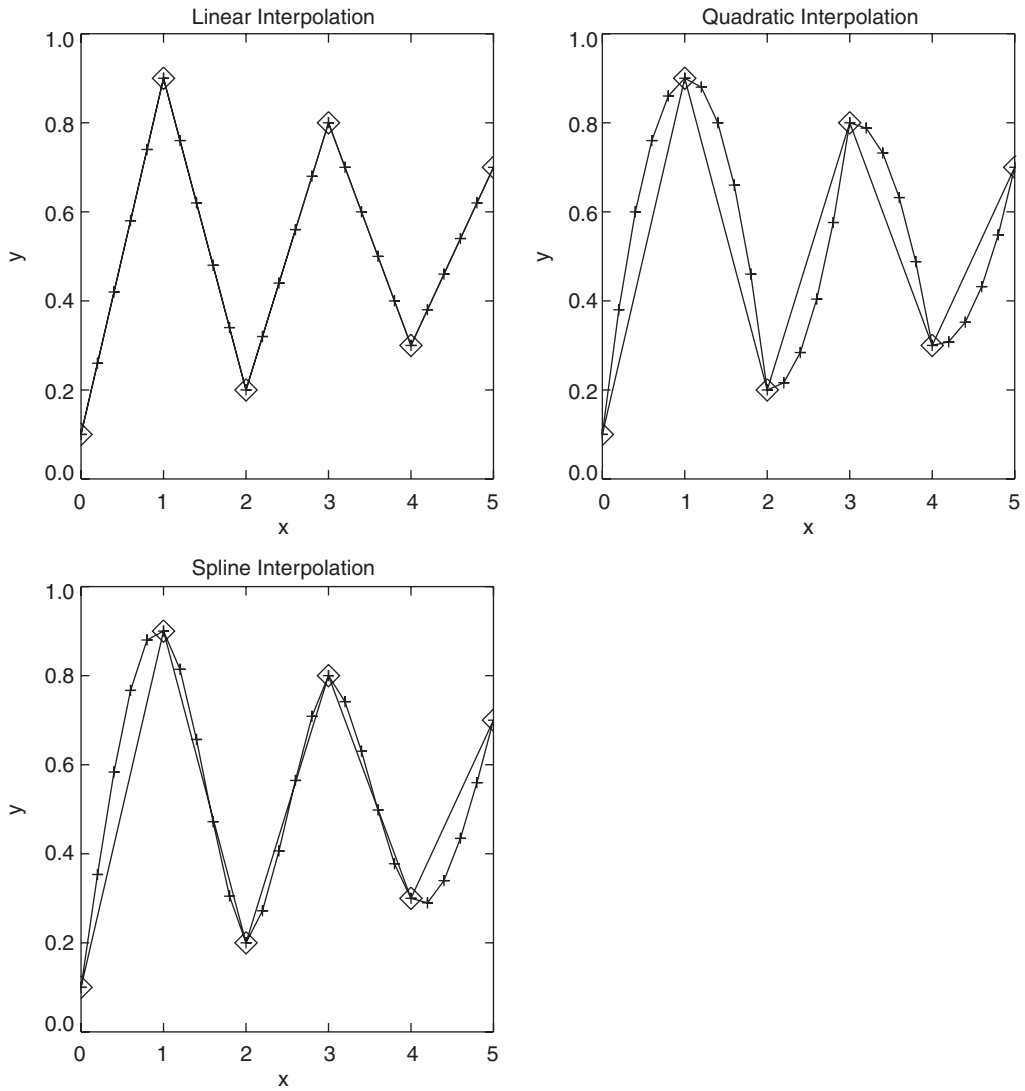

FIGURE 24.1 Examples of 1-D interpolation using linear interpolation (top left), quadratic interpolation (top right), and spline interpolation (bottom left). (INTERPOLATE1)

This example starts by creating a regularly spaced, independent coordinate x and an oscillatory set of dependent values y. The coordinates of the tabulated points do not have to be regularly spaced, but they must be monotonic (that is, in either increasing or decreasing order of x). The variable xx contains the coordinates of the points that we want to interpolate *to*. These points do not need to be monotonic. The interpolated values (yy) are computed using the INTERPOL function. By default, INTERPOL uses linear interpolation. Finally, the original points (x, y) are plotted and the interpolated points (xx, yy) are overplotted. The resulting graph is the upper left panel of Figure 24.1. The original data points are indicated by diamonds, the interpolated values by pluses. As expected for a piecewise linear interpolating function, the interpolated values lie on straight lines connecting the tabulated points.

To use a quadratic interpolating function, add the QUADRATIC keyword:

```
IDL> yy = interpol(y, x, xx, /quadratic)
IDL> plot, x, y, psym = -4, symsize = 2
IDL> oplot, xx, yy, psym = -1
```

The result is plotted in the upper right panel of Figure 24.1. Because quadratic interpolation requires *three* data points to construct the pieces of the interpolating function, there are two possible choices for the points to be used to interpolate each segment. Either choice will be *asymmetric*. In part due to this asymmetry, interpolating functions of *odd* order are usually preferred (linear, cubic, etc.). In this case, you can see that although the interpolating function passes through the tabulated points, it has kinks at the tabulated points and looks obviously different on either side of those points.

Splines are interpolating functions that are specifically designed to be smooth. Setting the SPLINE keyword tells INTERPOL to use cubic splines, which ensures that the interpolating function and its first and second derivatives are continuous everywhere, including the tabulated points.

```
IDL> yy = interpol(y, x, xx, /spline)
IDL> plot, x, y, psym = -4, symsize = 2
IDL> oplot, xx, yy, psym = -1
```

The resulting interpolated points are shown in the lower left panel of Figure 24.1. Note that the extrema of the interpolated values do not coincide with the tabulated points.

As you can see, interpolation schemes of different order have different characteristics that need to be taken into account when selecting an interpolation method. Higher order does not necessarily mean better!

24.4 Bilinear Interpolation

IDL includes two primary functions for doing two-dimensional interpolation. The simpler of the two is BILINEAR, which, as the name suggests, performs

bilinear interpolation. Bilinear interpolation is often used to interpolate two-dimensional gridded data between similar data grids (from the corners of a rectangular grid to the centers of the grid boxes, for example) or when a fast, simple interpolation scheme is sufficient.

The concept of bilinear interpolation is illustrated in Figure 24.2. Tabulated values of a function z are assumed to be available on a two-dimensional grid, indicated by black dots. The grid does not need to be regular (evenly spaced), but the grid lines do need to be perpendicular; that is, the x-coordinates of the grid points depend only on i, and the y-coordinates depend only on j.

The desired quantity is the value $\hat{z}$ at the point $(\hat{x}, \hat{y})$, which is indicated by the red circle. Applying the ideas of linear interpolation to this two-dimensional problem suggests two possible approaches. One is to interpolate first in the x-direction to get values at the locations marked by the filled red squares. Then interpolate in the y-direction to get $\hat{z}$. The second approach would be to interpolate first in the y-direction to get values at the locations marked by the open red squares. Then interpolate in the x-direction to get $\hat{z}$. This ambiguity suggests that one might get different answers depending on the order in which the calculation is done. In fact, comparing the two approaches reveals that, due to the linearity of the method, the two approaches give the same answer. (The algorithm is

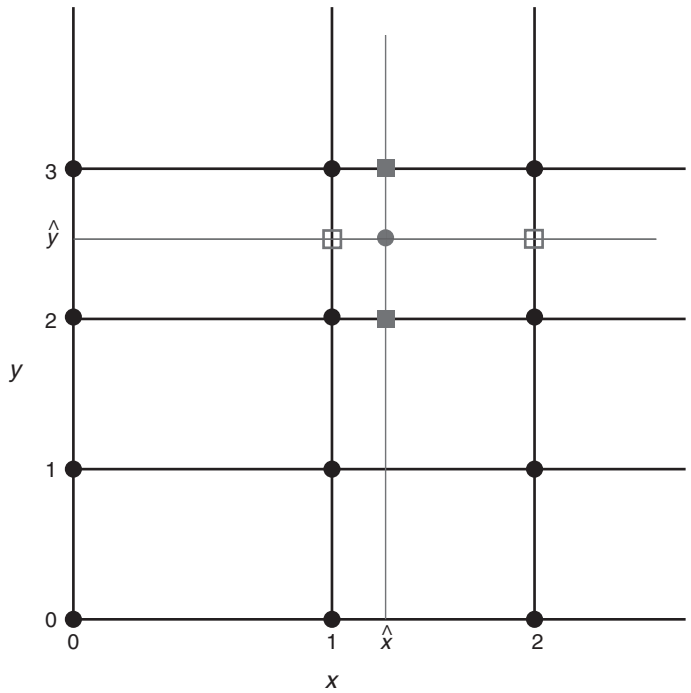

FIGURE 24.2 Schematic illustrating the concept of bilinear interpolation. Also see the color version of this figure in the color plates. (Not IDL)

usually implemented by computing *weights* w so that, when $\hat{x}$ lies between x_i and x_{i+1} and $\hat{y}$ lies between y_j and y_{j+1}, the result can be written $\hat{z} = w_{i,j}\, z_{i,j} + w_{i+1,j}\, z_{i+1,j} + w_{i,j+1}\, z_{i,j+1} + w_{i+1,j+1}\, z_{i+1,j+1}$. The weights depend on $\hat{x}$ and $\hat{y}$.)

BILINEAR requires only three arguments and has no keywords. The user need only supply the 2-D array of tabulated data and the coordinates of the output grid ($\hat{x}$'s and $\hat{y}$'s). Here is a simple example that interpolates coarsely gridded values of the function $z(x, y) = \sin(\pi x)\sin(\pi y)$ to a finer grid. The original coordinates x and y both range from 0 to 1.

```
IDL> WINDOW, XSIZE = 600, YSIZE = 600
IDL> !P.MULTI = [0, 2, 2]
IDL> x_lo  = FINDGEN(5)/4
IDL> y_lo  = FINDGEN(5)/4
IDL> z_lo  = SIN(!PI*x_lo) # SIN(!PI*y_lo)
IDL> SURFACE, z_lo, x_lo, y_lo
```

The resulting surface plot is shown in the upper left panel of Figure 24.3. For comparison, a higher-resolution version of data is plotted in the upper right panel of Figure 24.3.

```
IDL> x_hi  = FINDGEN(17)/16
IDL> y_hi  = FINDGEN(17)/16
IDL> z_hi  = SIN(!PI*x_hi) # SIN(!PI*y_hi)
IDL> SURFACE, z_hi, x_hi, y_hi
```

The higher-resolution grid gives a much smoother picture of the underlying function. Finally, the low-resolution data are interpolated to the high-resolution grid by using BILINEAR. The coordinates used by BILINEAR are *grid coordinates*, which are based on the indices of the grid points. In this example, the grid coordinates range from 0 to 4 in both directions. Unlike grid *indices*, which are integers, the grid coordinates are floating-point values. In Figure 24.2, $\hat{x} \approx 1.25$, while $\hat{y} \approx 2.5$. The user must provide the grid coordinates to BILINEAR. BILINEAR computes the interpolated values, which are returned as a 2-D array.

```
IDL> z_int = BILINEAR(z_lo, 4*x_hi, 4*y_hi)
IDL> SURFACE, z_int, x_hi, y_hi
```

The result is shown in the lower left panel of Figure 24.3. As can be seen in the figure, there is a noticeable difference between the interpolated values and the high-resolution values. Because the sine function is a complex curve, the bilinear interpolating function cannot fully capture its curvature. As a result, the interpolated values have "facets" between the tabulated data points. This is a reminder that interpolation does not magically fill in between known data points; it only provides an estimate of the unknown values.

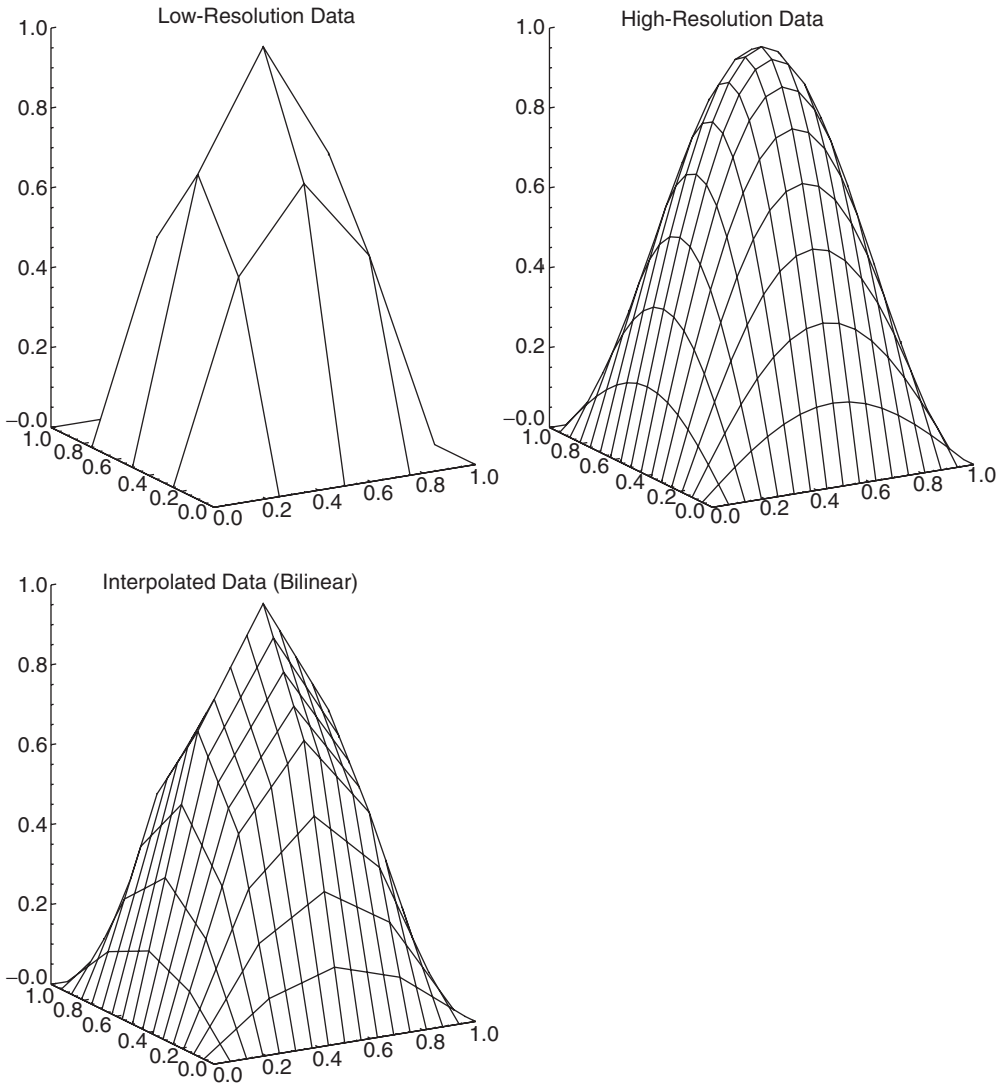

FIGURE 24.3 Examples of bilinear (2-D) interpolation. Original low-resolution function (top left), high-resolution version of original function (top right), original function interpolated to high-resolution grid (bottom left). (`BILINEAR1`)

24.5 Higher Dimensions

The IDL function `INTERPOLATE` will do one-, two-, and three-dimensional linear interpolation. It will also do *cubic convolution* on two-dimensional arrays. If you need to interpolate data with more than three dimensions, you may be able to use the built-in IDL functions on one or two dimensions at a time, or you may be forced to develop your own interpolation procedure. There are a great many different interpolation schemes that are not included in the IDL built-in functions. Before writing your own procedure, be sure to search the publicly available IDL libraries. Someone may have already done the work for you!

24.6 Irregular Grids

IDL has several built-in tools for dealing with irregularly gridded data. Data can be considered to be irregularly gridded if they do not fit naturally into standard rectangular data arrays. An example of irregularly gridded data would be temperatures at major cities. The locations of cities do not fall onto a rectangular grid.

One useful approach to analyzing and displaying irregularly gridded data is *triangulation*. When a data set is triangulated, a network or mesh of triangles is constructed with the data points at the vertices of the triangles. The mesh of triangles defines a piecewise-planar interpolating function; that is, each triangle is a piece of a plane surface. Note that the mathematical form of the triangular surfaces (flat planes) is different from the bilinear functions used for interpolating rectangularly gridded data.[1]

Given the x and y coordinates of a set of irregularly distributed data points, the IDL procedure `TRIANGULATE` will construct a triangular mesh from those points (known as a Delaunay triangulation) and return a list of the indices of the vertices of each triangle. Constructing the triangular mesh requires only a single IDL command, but plotting the results is slightly more complicated than some other types of plots. Therefore, this process is demonstrated using the IDL script below. (The script is available in the file `triangulate_script.pro` in the script's directory.) The graphs produced by the script are shown in Figure 24.4.

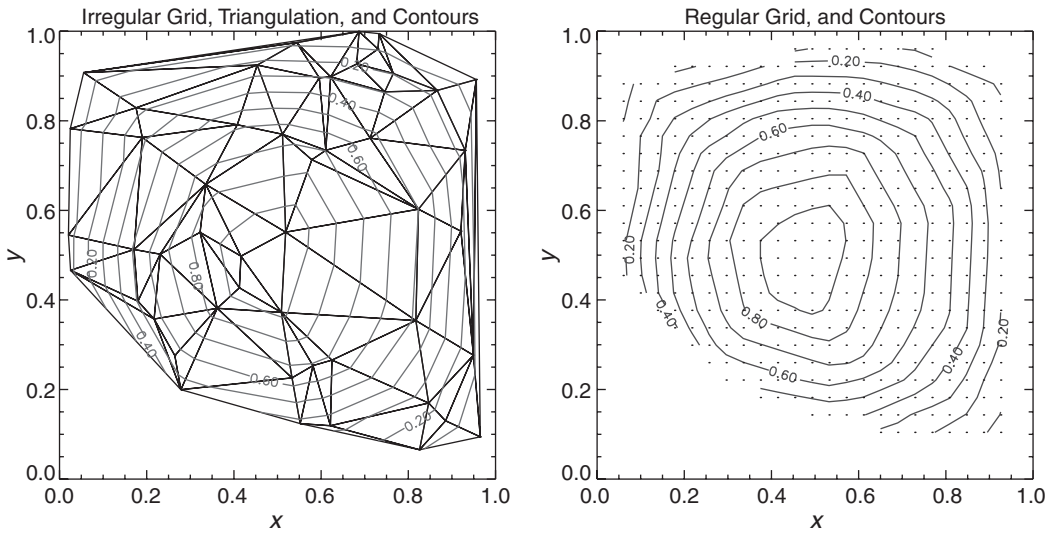

FIGURE 24.4 Examples of a 2-D triangular mesh created from irregularly gridded data by `TRIANGULATE` (left panel) and the data interpolated to a regular rectangular grid (right panel). Also see the color version of this figure in the color plates section. (`TRIANGULATE_PS`)

1 A rectangular grid could be converted to a triangular grid by drawing a diagonal through each rectangle of the grid. The triangles could then be used to construct an interpolating function for the data. Depending on which diagonal is chosen, however, the resulting triangles are generally different, which introduces ambiguity into the problem.

```
WINDOW, XSIZE = 800, YSIZE = 400            ;Open graphics window
!P.MULTI      = [0, 2, 1]                   ;Two graphics panes


;  PART 1 - Create irregular grid and display triangulation

n    = 50                                   ;Number of random points
seed = 47                                   ;Make result reproducible
x    = RANDOMU(seed, n)                     ;x-coords of irregular grid
y    = RANDOMU(seed, n)                     ;y-coords of irregular grid
z    = SIN(!PI*x)*SIN(!PI*y)                ;Compute dependent variable

TRIANGULATE, x, y, tri                      ;Compute triangulation
ntri = (SIZE(tri))[2]                       ;Number of triangles

PLOT, x, y, PSYM = 3, $                     ;Plot data points
   TITLE  = 'Irregular Grid and Triangulation', $
   XTITLE = 'x', $
   YTITLE = 'y'
FOR i = 0, ntri-1 DO $                      ;Draw each triangle
   PLOTS, [x[tri[*,i]], x[tri[0,i]]], $
          [y[tri[*,i]], y[tri[0,i]]]
CONTOUR, z, x, y, TRIANGULATION = tri, $    ;Draw contours using triangles
   /OVERPLOT, /FOLLOW, $
   LEVELS = 0.1*FINDGEN(11), $
   COLOR  = COLOR_24('red')


;  PART 2 - Interpolate data to a regular grid and plot using CONTOUR

nx = 25                                     ;x-resolution of regular grid
ny = 25                                     ;y-resolution of regular grid
zz = TRIGRID(x, y, z, tri, $                ;Interpolate to regular grid
        NX    = nx, NY    = ny, $           ;Resolution of output grid
        XGRID = xx, YGRID = yy, $           ;Coordinates of output grid
        MISSING = !VALUES.F_NAN)            ;Points outside triangles are
                                             set to NaN
CONTOUR, zz, xx, yy, /FOLLOW, $             ;Contour data on regular grid
   C_COLOR = COLOR_24('blue'), $
   LEVELS  = 0.1*FINDGEN(11), $
   TITLE   = 'Regular Grid and Contours', $
   XTITLE  = 'x', $
   YTITLE  = 'y'

xg = REBIN(           xx, nx, ny, /SAMPLE)  ;Make xx into 2-D grid
yg = REBIN(TRANSPOSE(yy), nx, ny, /SAMPLE)  ;Make yy into 2-D grid
```

```
i  = WHERE(FINITE(zz))                    ;Find points within triangulation
PLOTS, xg[i], yg[i], PSYM = 3             ;Plot grid points within
                                           triangulation


!P.MULTI = 0                              ;Restore !P.MULTI
```

The first two lines of the script open a graphics window for two plots.

Next, the script creates an irregular grid of 50 data points by using the RANDOMU function to generate random x and y coordinates between 0 and 1. For the dependent variable z we use the same function as in the previous examples, $z(x, y) = \sin(\pi x)\sin(\pi y)$. The triangular mesh is computed using the TRIANGULATE procedure. The list of the indices of the vertices of the triangles is returned in the array tri, which is dimensioned $3 \times$ ntri, where ntri is the number of triangles needed to create the mesh. We use the SIZE function to get the number of triangles from the dimensions of tri.

Next, the data points are plotted (do not connect the dots!), and then, for each triangle, the three sides are drawn using the coordinates of the vertices of the triangles. Note that some triangles are nearly equilateral, whereas others are long and thin.

Given the irregularly gridded data and the list of triangles, the CONTOUR procedure will draw contour lines. These are drawn in red on top of the triangular mesh. Notice that the contours are straight lines within each triangle. This results from the fact that the contour segments are straight lines defined by the intersection of each triangle and the surfaces $z = \{0.0, 0.1, 0.2, \ldots, 1.0\}$. As you can see, although the function z is symmetric around the center of the plot box, the contours are not. Also, sizable parts of the box have no data points at all. This indicates that this set of 50 randomly distributed points is not sufficient to characterize this function well. Setting the IRREGULAR keyword to CONTOUR is equivalent to calling TRIANGULATE and then CONTOUR with the TRIANGULATION keyword.

If the only use of the data is to display contour plots, then the steps above are sufficient. In some cases, however, it is useful to interpolate the irregularly gridded data onto a regular grid. This can be done by using the TRIGRID function, which is demonstrated in the second part of the script.

The properties of the regular output grid can be specified by using various keywords of the TRIGRID function. Here we specify that the output grid be dimensioned 25×25. By default, the grid is created so that its rectangular border just includes all of the points of the mesh. The coordinates of the grid points are returned in the arrays xx and yy. Points that fall outside the boundary of the triangular mesh are set to NaN. Points inside are interpolated using the triangular mesh computed earlier by TRIANGULATE. If desired, points outside the mesh can be estimated by extrapolation, but the results are often unsatisfactory. The interpolated values on the regular grid are returned in the array zz.

The regularly gridded interpolated values are plotted in blue using a standard call to CONTOUR *without* the TRIANGULATE keyword. Finally, the locations of the regular grid points that fall within the triangular mesh (points with values that are not NaN) are drawn. Because contours are drawn differently on the irregular and regular grids, the two sets of contours are very similar, but not identical. You can see this by modifying the script `triangulate.pro` to overplot the two sets of contours on the same graph.

24.7 Summary

This chapter covers the basics of interpolation using the INTERPOL and BILINEAR functions. Displaying irregularly gridded data by using a triangular mesh, and interpolating to a regular grid are demonstrated using the TRIANGULATE and TRIGRID functions.